



A LINGUAGEM DE PROGRAMAÇÃO C

```
1 #include <goodluck.h>
2
3 int main (void){
4
5     printf("Guilherme Pina Cardim");
6     printf("\n\n");
7     printf("Ruben de Best")
8     printf("\n\n");
9     printf("Prof. Dr. Rogério Eduardo Garcia");
10
11     return(10);
12
13 }
```



A Linguagem C

“C é uma linguagem de programação de finalidade geral que permite economia de expressão, modernos fluxos de controle e estruturas de dados e um rico conjunto de operadores. Sua falta de restrições e sua generalidade tornam-na mais conveniente e eficaz para muitas tarefas do que linguagens supostamente mais poderosas.”

Ritchie e Kernighan.



Tópicos

- Histórico;
- Classificação;
- A que área se destina;
- Critérios de Avaliação;
- Outras Características;
- Mapa de Memória de um Programa C;
- Identificadores;
- Tipos de Dados e Tamanhos;



Tópicos (...)

- Variáveis;
- Vinculações;
- Variáveis Estáticas;
- Variáveis Dinâmicas de Pilha;
- Variáveis Dinâmicas de Monte Explícitas;
- Escopo;
- Variável *Extern*;
- Register;



Tópicos (...)

- Char;
- Strings;
- Struct;
- Union;
- Tipo Ordinal;
- Matrizes;
- Estruturas de Controle;
- Processo de Compilação;
- Bibliografia.



Histórico

Desenvolvida por Dennis Ritchie nos laboratórios da BELL (AT&T Bell Labs) por volta dos anos 70 (1969-1973), visando a implementação do Unix. Foi derivada da linguagem B, a qual se originou a partir da linguagem BCPL (Linguagem de Programação Básica Compilada). Descrito por Ritchie e Kernighan em 1978, porém somente em 1983 o ANSI (American National Standards Institute) definiu um padrão para a linguagem C.



Classificação

- Médio Nível;
 - Combina elementos de alto nível com funcionalidades de baixo nível da linguagem Assembly.
- Terceira Geração;
 - Linguagem orientada ao usuário;
 - Subclassificação: Procedimental.
- Paradigma Imperativo Estruturado.
 - Centrado no conceito de um estado e ações que o manipulam;
 - Instruções agrupadas em blocos.



A que área se destina?

A linguagem C não se destina a nenhuma área em específico. É uma linguagem de propósito geral. Ela é utilizada para diversos fins, inclusive para escrever compiladores de outras linguagens. É uma linguagem potente e flexível.





Critérios de Avaliação

- **Legibilidade;**
- **Capacidade de Escrita;**
- **Confiabilidade;**
- **Custo.**





Legibilidade

- Simplicidade Global: número de operadores e palavras reservadas é pequeno. Sobrecarga de operadores e multiplicidade de recursos.
 - Exemplo: Incrementar uma variável inteira:
 - `cont = cont + 1;`
 - `cont += 1;`
 - `cont++;`
 - `++cont;`



Legibilidade (...)

- Ortogonalidade: Existe falta de ortogonalidade em alguns aspectos de C.
 - Um registro pode ser retornado de uma função, mas um *array* não.
 - Parâmetros podem ser passados por valor, exceto *array* que deve ser passado por referência.
 - Um elemento de uma estrutura não pode ser *void*, ou uma estrutura do mesmo tipo.
 - Um elemento de um *array* não pode ser *void* ou uma função.



Legibilidade (...)

- Instruções de Controle: Programação estruturada (*while*, *for*, ...) reduzindo o uso do *goto* e consequentemente melhorando a legibilidade.
- Tipos de Dados e Estruturas: Possui facilidades para definir tipos e estruturas de dados adequadas para determinado problema.



Legibilidade (...)

- Sintaxe: três fatores de sintaxe:
 - Formas de Identificadores: o limite para identificadores em C não é pequeno, permitindo expressar o nome do identificador mais adequadamente.
 - Palavras reservadas: C não permite utilizar palavras reservadas como variáveis.



Legibilidade (...)

- Sintaxe: três fatores de sintaxe:
 - Forma e significado: Instruções que a aparência indica a finalidade. Em C: -> (Ponteiro)

Violação em C: A palavra reservada *static* possui diferentes significados:

- Definição de variável **dentro** da função: variável é criada no momento da compilação;
- Definição de variável **fora** de funções: é visível apenas no arquivo onde ocorre a definição.

Em relação a variáveis *static* globais, Schildt (1996) afirma que esta é reconhecida apenas no arquivo no qual a mesma foi declarada.



Legibilidade (...)

- Violação em C do *static*:

```
//arquivo c.c
```

```
static int i=1;
```

Saída:

Valor de i[1]

```
#include <stdio.h>
```

```
#include "c.c"
```

```
extern int i;
```

```
int main(void)
```

```
{
```

```
    printf("Valor de i [%d]\n",i);
```

```
    return 0;
```

```
}
```

Testes efetuados com os
compiladores gcc 4.6.3,
Borland C++ 5.6 e Turbo C
3.0.

A ideia descrita pela literatura de que uma variável *static* não é vista por outros arquivos não foi comprovada em nenhum dos compiladores testados (gcc 4.6.3; Borland C++ 5.6 e Turbo C 3.0). Todos estes compiladores permitiram que a variável *static* declarada no arquivo "c.c" fosse vista e utilizada pelo arquivo principal.



Capacidade de Escrita

- Simplicidade e Ortogonalidade: demasiada ortogonalidade pode prejudicar a capacidade de escrita. A linguagem C possui poucos construtores e um pequeno conjunto de regras para combiná-los.
- Suporte para Abstração: C permite a abstração de problemas reais com a utilização de tipos de dados existentes ou definidos juntamente com a definição de funções e seus comportamentos diante da abstração proposta.



Capacidade de Escrita (...)

- Expressividade: C possui instruções expressivas, como:
 - **cont++**; ao invés de **cont = cont + 1**;
 - **cont -= x**; ao invés de **cont = cont - x**;
 - **for(i=0; i<100; i++){ }** para laços de contagem ao invés de **while**.



Confiabilidade

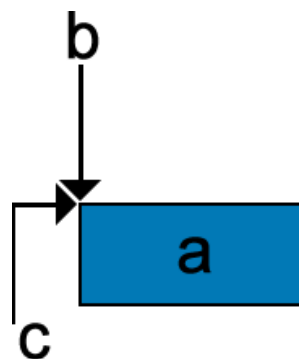
- Verificações de Tipos: em C ANSI os tipos dos parâmetros passados para uma função não eram verificados de acordo com os tipos definidos pela função. Atualmente os compiladores já fazem esta verificação.
- Manipulação de exceções: praticamente não existe manipulação de exceções ocorridas em tempo de execução em C. Inclui facilidades que permitem a implementação de uma funcionalidade similar.



Confiabilidade (...)

- Aliasing: em C pode-se utilizar o conceito de *aliasing* facilmente pela criação de dois ponteiros apontando para o mesmo local de memória, ou pelo uso do tipo *union*.

```
1  #include <stdio.h>
2
3  int main(void) {
4      int a = 10;
5      int *b, *c;
6      b = &a;
7      c = &a;
8      return(0);
9  }
```



- Legibilidade e Capacidade de Escrita: Descritas e especificadas anteriormente.



Custo

- Treinamento;
- Escrita;
- Compilação;
- Execução;
- Sistemas de Implementação da linguagem;
- Confiabilidade;
- Manutenção.



Critérios de Avaliação (...)

Característica	Legibilidade	Capacidade de Escrita	Confiabilidade
Simplicidade/Ortogonalidade	X	X	X
Estruturas de controle	X	X	X
Tipos de dados e estruturas	X	X	X
Projeto de sintaxe	X	X	X
Suporte para abstração		X	X
Expressividade		X	X
Verificação de tipos			X
Manipulação de exceções			X
Apelido restrito			X



Outras Características

- **Desenvolvimento em Unix:** é a linguagem oficial deste sistema operacional;
- **Flexível:** de propósito geral, não é limitada para nenhum tipo de aplicação;
- **Eficiente:** possuindo funcionalidades de baixo nível, consegue obter performances semelhantes ao Assembly;
- **Simples:** com um número pequeno de palavras reservadas, tipos de dados e operadores;



Outras Características (...)

- **Popular:** internacionalmente conhecida e utilizada. Além de muito bem documentada, existem compiladores para praticamente todas as plataformas e arquiteturas de computadores;
- **Portabilidade:** sendo definida por uma padrão ANSI, de modo geral o código escrito em uma máquina pode ser transportado e compilado em outra máquina sem maiores complicações.

Mapa de Memória de um Programa C





Identificadores

- Identificadores devem iniciar com uma letra ou um *underscore*, sendo seguidos por mais letras, *underscores*, ou números.
- Podem ter no máximo 31 caracteres.
- Linguagem *case-sensitive*.
- Palavras reservadas não podem ser utilizadas como outros identificadores.



Identificadores

Relação de palavras reservadas no padrão ANSI:

<i>auto</i>	<i>double</i>	<i>int</i>	<i>struct</i>
<i>break</i>	<i>else</i>	<i>long</i>	<i>switch</i>
<i>case</i>	<i>enum</i>	<i>register</i>	<i>typedef</i>
<i>char</i>	<i>extern</i>	<i>return</i>	<i>union</i>
<i>const</i>	<i>float</i>	<i>short</i>	<i>unsigned</i>
<i>continue</i>	<i>for</i>	<i>signed</i>	<i>void</i>
<i>default</i>	<i>goto</i>	<i>sizeof</i>	<i>volatile</i>
<i>do</i>	<i>if</i>	<i>static</i>	<i>while</i>



Tipos de Dados e Tamanhos

- Há poucos tipos primitivos de dados em C.
 - *void; char; int; float; double.*
- Podem ser aplicados qualificadores a estes tipos primitivos, disponibilizando tamanhos diferentes quando isto for necessário.
 - *short; long; signed; unsigned.*



Variáveis

- Variáveis são abstrações de uma célula de memória.
- Aumentam muito a legibilidade e facilitam a manutenção do código.
- Caracterizados pela sêxtupla:
 - Nome, endereço, valor, tipo, tempo de vida e escopo.
- É necessária a declaração de todas as variáveis.

Vinculação de Atributos a Variáveis



- Relembrando
 - Uma vinculação é **estática** se ela ocorre pela primeira vez antes do tempo de execução e permanece intocada ao longo da execução do programa.
 - Uma vinculação é **dinâmica** se ela ocorre pela primeira vez durante a execução do programa ou se ela pode ser alterada durante o tempo de execução.



Vinculações de Tipos

- Em C a vinculação de tipos é estática e é feita uma declaração explícita, ou seja, existe uma sentença no programa que lista os nomes das variáveis e seu tipo.



```
int x,y,z;  
char a,b,c;
```

Vinculações de Armazenamento e Tempo de Vida



- O **tempo de vida** de uma variável é o tempo durante o qual ela está vinculada a uma posição específica da memória.
 - Variáveis estáticas.
 - Variáveis dinâmicas de pilha.
 - Variáveis dinâmicas de monte explícitas.

Obs: Variáveis dinâmicas de monte implícitas não existem em C.

Variáveis estáticas



- Vantagens
 - Endereçamento direto.
 - Sensíveis ao histórico.
- Desvantagens
 - Não podem ser utilizadas em recursão.
 - Redução da flexibilidade.
 - Desperdício de memória.

```
#include <stdio.h>

void funcao_a()
{
    static int a,b,c;
}

int main (void)
{
    funcao_a();
    return 0;
}
```

Teste efetuado com o compilador gcc 4.6.3

Variáveis dinâmicas de pilha



- Vantagens
 - Permitem recursão.
 - Melhor utilização de memória.
- Desvantagens
 - Não sensíveis ao histórico.
 - Endereçamento indireto.
 - Sobrecarga em tempo de execução de alocação e liberação.

```
#include <stdio.h>

void funcao_a()
{
    int a,b,c;
}

int main (void)
{
    funcao_a();
    return 0;
}
```

Teste efetuado com o compilador gcc 4.6.3

Variáveis Dinâmicas de Monte Explícitas



- Células de memória alocadas e liberadas por instruções explícitas tem tempo de execução pelo programador.
- Necessidade de utilização de ponteiros.
- Vantagens
 - Armazenamento dinâmico
 - Uso otimizado de memória.
- Desvantagens
 - Em C existe a necessidade de liberação da memória pelo programador.
 - Ineficiente.
 - Altamente sujeito a erros.

Variáveis Dinâmicas de Monte Explícitas



```
#include <stdio.h>
#include <stdlib.h>

typedef struct _node {
    struct _node * prox;
    int valor;
} NODE;

int main (void) {
    NODE * no = (NODE *) malloc (sizeof(NODE)) ;
    free(no) ;
    return 0;
}
```

Escopo



- O escopo de uma variável é a faixa de sentenças nas quais ela é visível.
- Variáveis podem ser:
 - Locais
 - Globais
- Em C o escopo das variáveis é estático, ou seja, pode ser definido antes da execução.

Escopo Local



- São declaradas em funções ou em blocos.
- Podem ser acessadas dentro da função ou do bloco onde foram declaradas.

```
#include <stdio.h>

int main(void) {
    int i = 2;
    printf("%d\n",i); //Imprime 2
    {
        int i = 0;
        printf("%d\n",i); //Imprime 0
    }
    return 0;
}
```

Teste efetuado com o compilador gcc 4.6.3

Escopo Global



- São reconhecidas em todo o programa.
- Se uma variável local utilizar o mesmo nome, a variável global não poderá ser referenciada.

```
#include <stdio.h>

int i = 1;

int main(void) {
    printf("%d\n",i); //Imprime 1
    {
        int i = 0;
        printf("%d\n",i); //Imprime 0
    }
    return 0;
}
```

Teste efetuado com o compilador gcc 4.6.3

Variável extern



- Apenas o tipo e o nome da variável são definidos, e se referem a uma variável declarada em outro arquivo. Na compilação esta dependência é resolvida.
- Quando o compilador encontrar duas variáveis possíveis (com mesmo nome) para *linkar* a uma variável definida como *extern* o compilador avisa o usuário de uma dupla declaração da variável.

Escopo / extern



```
#include <stdio.h>
#include "escopo.c"

int i = 1;
extern int j;

int main(void) {
    printf("%d %d\n", i, j);
    {
        int i = 2, j = 3;
        printf("%d %d\n", i, j);
        {
            int i = 0;
            printf("%d %d\n", i, j);
        }
        printf("%d %d\n", i, j);
    }
    printf("%d %d\n", i, j);
    return 0;
}
```

```
/* escopo.c */
#include <stdio.h>

int j = 2;
```

Saída:

```
1 2
2 3
0 3
2 3
1 2
```


register



- Utilizados para aumento de desempenho.
- Só podem ser declarados localmente.
- Armazenamento em registradores da CPU ao invés de na memória.



register



```
#include <stdio.h>

#define SIZE 100000

int main (void)
{
    register int i,j;
    for (i=0;i<SIZE;i++)
        for (j=0;j<SIZE;j++);

    return 0;
}
```

Teste efetuado com o compilador gcc 4.6.3

real	0m6.252s
user	0m6.252s
sys	0m0.000s

```
#include <stdio.h>

#define SIZE 100000

int main (void)
{
    int i,j;
    for (i=0;i<SIZE;i++)
        for (j=0;j<SIZE;j++);

    return 0;
}
```

Teste efetuado com o compilador gcc 4.6.3

real	0m20.130s
user	0m20.133s
sys	0m0.000s



char

- É um inteiro de 1 byte.
- Podemos fazer aritmética sobre char como fazemos sobre inteiros.
- Letras e símbolos são exibidos utilizando-se a codificação ASCII. Esta tabela pode ser encontrada em www.asciitable.com

char



```
#include <stdio.h>

int main (void)
{
    char c;

    for (c = 'A'; c <= 'Z'; c++)
        //Imprime de [A a] até [Z z]
        printf("[%c %c]\n", c, c+32);

    for (c=65; c<=90; c++)
        //Imprime de [A a] até [Z z]
        printf("[%c %c]\n", c, c+32);

    return 0;
}
```

char



```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    char c;
```

```
    for (c = 'A'; c <= 'Z'; c++);
```

```
    for (c=65;c<=90;c++);
```

```
    return 0;
```

```
}
```

Teste efetuado com o compilador gcc 4.6.3

```
main:
```

```
    pushl    %ebp
```

```
    movl     %esp, %ebp
```

```
    subl     $16, %esp
```

```
    movb     $65, -1(%ebp)
```

```
    jmp      .L2
```

```
.L3:
```

```
    addb     $1, -1(%ebp)
```

```
.L2:
```

```
    cmpb     $90, -1(%ebp)
```

```
    jle      .L3
```

```
    movb     $65, -1(%ebp)
```

```
    jmp      .L4
```

```
.L5:
```

```
    addb     $1, -1(%ebp)
```

```
.L4:
```

```
    cmpb     $90, -1(%ebp)
```

```
    jle      .L5
```

```
    movl     $0, %eax
```

```
    leave
```

```
    ret
```



Strings

- *String* não é um tipo primitivo do C.
- Pode ser implementado através de *array* de *char*.
- Existe a biblioteca `string.h` que é padrão ANSI e foi criada para fazer a manipulação desses *arrays* de *char*.



struct

- Uma *struct* consiste de uma lista de campos.
- O tamanho total de uma *struct* é calculado pela soma dos tamanhos dos seus campos . Em alguns compiladores existe a necessidade de um *padding* de controle. Este *padding* é feito para garantir o correto alinhamento.

struct



```
struct exemplo
{
    char a;
    int ia;

    char b;
    int ib;

    char c;
    int ic;

    char d;
    int id;
};
```

sizeof(struct exemplo)=32
sizeof(struct exemplo2)=20

Teste efetuado com o compilador gcc 4.6.3 e
Borland C++ 5.6

sizeof(struct exemplo)=12
sizeof(struct exemplo2)=12

Teste efetuado com o compilador Turbo C 3.0

```
struct exemplo2
{
    char a;
    char b;
    char c;
    char d;

    int ia;
    int ib;
    int ic;
    int id;
};
```

```
movb    $1, -72(%ebp)
movl    $100, -68(%ebp)
movb    $2, -64(%ebp)
movl    $200, -60(%ebp)
movb    $3, -56(%ebp)
movl    $300, -52(%ebp)
movb    $4, -48(%ebp)
movl    $400, -44(%ebp)

movb    $1, -40(%ebp)
movb    $2, -39(%ebp)
movb    $3, -38(%ebp)
movb    $4, -37(%ebp)
movl    $100, -36(%ebp)
movl    $200, -32(%ebp)
movl    $300, -28(%ebp)
movl    $400, -24(%ebp)
```


union



- Uma *union* é uma coleção de variáveis de tipos diferentes, sendo que só é possível guardar informações em uma variável por vez.
- Uma *union* tem o tamanho da maior variável nela presente.
- Melhor aproveitamento de memória.
- Dificuldade de utilização e altamente sujeito a erros.

union



Teste efetuado com o compilador gcc 4.6.3

```
#include <stdio.h>
#include <string.h>

union exemplo
{
    char c[8];
    int i[2];
    double d;
};

int main(void)
{
    union exemplo a;

    strcpy(a.c, "L.P.");

    printf("char[8] = %s\n", a.c); //Impressão correta
    printf("int [2] = %d %d\n", a.i[0], a.i[1]); //Impressão incorreta
    printf("double = %lf", a.d); //Impressão incorreta

    a.i[0] = 10; a.i[1] = 20;

    printf("char[8] = %s\n", a.c); //Impressão incorreta
    printf("int [2] = %d %d\n", a.i[0], a.i[1]); //Impressão correta
    printf("double = %lf", a.d); //Impressão incorreta

    a.d = 123.45;

    printf("char[8] = %s\n", a.c); //Impressão incorreta
    printf("int [2] = %d %d\n", a.i[0], a.i[1]); //Impressão incorreta
    printf("double = %lf", a.d); //Impressão correta

    printf("Sizeof: %d\n", sizeof(union exemplo)); //Tamanho 8 bytes
}
```



Tipo Ordinal

- Tipo enumerado:
 - Utilizado principalmente para aumentar legibilidade do código.
 - Exemplo:
 - `enum week {Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun};`
- Tipo sub-faixa não existe em C.
- Vantagem sobre *#define* é que enum tem escopo.



Matrizes

- Formato básico:
 - *tipo* array [dim], onde:
 - tipo: tipo de dado
 - array: nome do array
 - dim: dimensão do array. Deve ser inteiro.
- Alocação de um bloco contínuo de memória.
 - Tamanho (array) = $\text{sizeof}(\text{tipo}) * \text{dim}$



Matrizes

- Matriz estática
- Matriz dinâmica da pilha fixa
- Matriz dinâmica da pilha
- Matriz dinâmica do monte fixa





Matriz Estática

- Vinculação estática de índices.
- Alocação de armazenamento estática, feita antes da execução.
- Vantagem:
 - Eficiente: Nenhuma alocação ou liberação dinâmica é necessária.
- Desvantagens:
 - Armazenamento vinculado por toda a execução.



Matriz Estática

```
#include <stdio.h>

#define SIZE 100

int array_global[SIZE];

int main(void)
{
    static int array_static[SIZE];
    return(0);
}
```

Teste efetuado com o compilador gcc 4.6.3



Matriz Dinâmica da Pilha Fixa

- Vinculação estática de índices.
- Alocação de armazenamento feita em tempo de execução.
- Vantagens:
 - Uso de memória mais eficiente.
- Desvantagens:
 - Tempo necessário para a alocação e liberação.



Matriz Dinâmica da Pilha Fixa

```
#include <stdio.h>

#define SIZE 100

void funcao_a(void) {
    int array_local_a[SIZE];
}

void funcao_b(void) {
    int array_local_b[SIZE];
}

int main(void) {
    funcao_a();
    funcao_b();
    return(0);
}
```



Matriz Dinâmica da Pilha

- Faixas de índices e a alocação de armazenamento vinculadas em tempo de elaboração.
- Vantagens:
 - Utilização ainda mais eficiente de memória.
- Desvantagens:
 - Tempo necessário para a alocação e liberação.



Matriz Dinâmica da Pilha

```
#include <stdio.h>

void funcao_a(int size) {
    int array_local_a[size];
}

void funcao_b(int size) {
    int array_local_b[size];
}

int main(void) {
    int a,b;
    scanf("%d %d",&a, &b);
    funcao_a(a);
    funcao_b(b);
    return(0);
}
```

Este código executou perfeitamente no compilador gcc 4.6.3. No entanto os compiladores Borland C++ 5.6 e Turbo C 3.0 um erro é apresentado. Para estes compiladores é necessário uma constante, impedindo assim o uso de uma Matriz Dinâmica de Pilha.

Matriz Dinâmica de Monte Fixa



- Faixas de índices e a alocação de armazenamento vinculadas em tempo de elaboração.
- Diferentemente das anteriores, neste caso a memória alocada é do monte e não da pilha.
- Utilização das funções da biblioteca padrão `stdlib.h`:
 - `free`
 - `malloc`

Matriz Dinâmica de Monte Fixa



- Vantagens:
 - Flexibilidade – tamanho da matriz sempre se encaixa no problema.
- Desvantagens:
 - Tempo necessário para a alocação e liberação no monte é maior do que na pilha.
 - Necessidade de liberação manual de memória através do comando free.

Matriz Dinâmica de Monte Fixa



```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    scanf("%d", &size);
    int * array = (int *) malloc (sizeof(int) * size);
    free(array);
    return(0);
}
```

Teste efetuado com o compilador gcc 4.6.3

Matrizes e Aritmética de Ponteiros



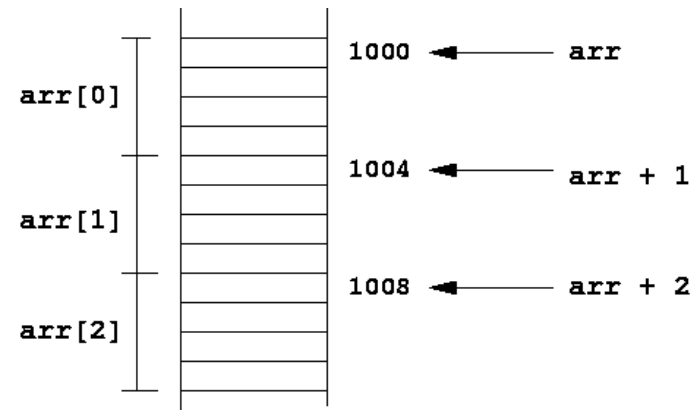
- `int arr[3]`

- Em C:

- `arr[i] = * (arr + i)`

- Em bytes:

- $\text{endereço}(\text{arr}[i]) = \text{endereço}(\text{arr}) + [\text{tamanho}(\text{int}) * i]$



Matrizes e Aritmética de Ponteiros



```
#include <stdio.h>
```

```
int main(void) {  
    short array[3];  
    array[0] = 10;  
    array[1] = 20;  
    array[2] = 30;  
    return(0);  
}
```

Teste efetuado com o compilador gcc 4.6.3

main:

```
    pushl    %ebp  
    movl     %esp, %ebp  
    subl     $16, %esp  
    movw     $10, -6(%ebp)  
    movw     $20, -4(%ebp)  
    movw     $30, -2(%ebp)  
    movl     $0, %eax  
    leave  
    ret
```

Guilherme P. Cardim e Ruben de Best

```
#include <stdio.h>
```

```
int main(void) {  
    int array[3];  
    array[0] = 10;  
    array[1] = 20;  
    array[2] = 30;  
    return(0);  
}
```

Teste efetuado com o compilador gcc 4.6.3

main:

```
    pushl    %ebp  
    movl     %esp, %ebp  
    subl     $16, %esp  
    movl     $10, -12(%ebp)  
    movl     $20, -8(%ebp)  
    movl     $30, -4(%ebp)  
    movl     $0, %eax  
    leave  
    ret
```




Matrizes Multidimensionais

- *Arrays* possuem apenas uma dimensão, mas elementos podem ser outros *arrays*.
- Formato básico:
 - *tipo* array [dim1]...[dimN], onde:
 - tipo: tipo de dado
 - array: nome do array
 - dim1,...,dimN: dimensões
 - Alocação de um bloco contínuo de memória.
 - Tamanho (*array*) = sizeof(tipo) * dim1 * ... * dimN

Matrizes Multidimensionais (...)



- `int arr[numLinhas][numColunas]`
 - Em C:
 - `arr[i][j] = * (arr + i * numColunas + j)`
 - Em bytes:
 - `endereço(arr[i][j]) = endereço(arr) + [sizeof (int) * i * numColunas + j * sizeof (int)]`



Matrizes Multidimensionais (...)



```
#include <stdio.h>
int main(void)
{
    int array[3][3];

    array[0][0] = 0;
    array[0][1] = 1;
    array[0][2] = 2;

    array[1][0] = 10;
    array[1][1] = 11;
    array[1][2] = 12;

    array[2][0] = 20;
    array[2][1] = 21;
    array[2][2] = 22;

    return (0);
}
```

```
main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $48, %esp
    movl     $0, -36(%ebp)
    movl     $1, -32(%ebp)
    movl     $2, -28(%ebp)
    movl     $10, -24(%ebp)
    movl     $11, -20(%ebp)
    movl     $12, -16(%ebp)
    movl     $20, -12(%ebp)
    movl     $21, -8(%ebp)
    movl     $22, -4(%ebp)
    movl     $0, %eax
    leave
    ret
```



Estruturas de Controle

- Testes:

- *if*:

```
if (condição) {  
    ... /* bloco a ser executado se a condição for verdadeira */  
}  
else {  
    ... /* bloco a ser executado se a condição for falsa */  
}
```

- *Operador Ternário*: `condição ? valorSeVerdadeira : valorSeFalsa`

- *switch*:

```
switch (expressão) {  
    case valor1:  
        instruções;  
        break;  
    case valor2:  
        instruções;  
        break;  
    ...  
    default:  
        instruções;  
}
```



Estruturas de Controle (...)

- Loops:

- *while:*

```
while (condição) {  
    ...  
}
```

- *do_while:*

```
do {  
    ...  
} while (condição);
```

- *for:*

```
for (inicialização; condição; incremento) {  
    instruções;  
}
```

=

```
inicialização;  
while (condição) {  
    instruções;  
    incremento;  
}
```



Estruturas de Controle (...)

- Comandos de Desvio:
 - *return*: encerra o fluxo em uma função e retorna para o local onde foi chamada;
 - *exit*: encerra o fluxo do programa, ou seja, finaliza o programa imediatamente;





Estruturas de Controle (...)

- Comandos de Desvio:
 - *break*: Finalizar um *case* na decisão de um *switch*, ou interromper imediatamente um loop.
 - *continue*: salta imediatamente para a próxima iteração do loop.


```
int main (void)
{
    int a, b;
    while ( a > 10)
    {
        if (a > b)
            break;
        b = 5;
    }
    return 0;
}
```

```
int main (void)
{
    int a, b;
    while ( a > 10)
    {
        if (a > b)
            continue;
        b = 5;
    }
    return 0;
}
```



Estruturas de Controle (...)

- Comandos de Desvio:
 - *goto*: é um comando de salto incondicional. Realiza um salto para um local especificador por um rótulo.



```
int main (void)
{
    int a, b;
    while ( a > 10)
    {
        if (a > b)
            goto nome;
        b = 5;
    }
    nome:
    return 0;
}
```

Guilherme P. Cardim e Ruben de Best



Estruturas de Controle (...)

- Comandos de Desvio: *break* X *continue* X *goto*

```
main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    jmp      .L2
.L5:
    movl     -4(%ebp), %eax
    cmpl     -8(%ebp), %eax
    jg       .L7
.L3:
    movl     $5, -8(%ebp)
.L2:
    cmpl     $10, -4(%ebp)
    jg       .L5
    jmp      .L4
.L7:
    nop
.L4:
    movl     $0, %eax
    leave
    ret
```

```
main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    jmp      .L2
.L4:
    movl     -4(%ebp), %eax
    cmpl     -8(%ebp), %eax
    jg       .L6
.L3:
    movl     $5, -8(%ebp)
    jmp      .L2
.L6:
    nop
.L2:
    cmpl     $10, -4(%ebp)
    jg       .L4
    movl     $0, %eax
    leave
    ret
```

Guilherme R. Cardim e Ruben de Best

```
main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    jmp      .L2
.L5:
    movl     -4(%ebp), %eax
    cmpl     -8(%ebp), %eax
    jg       .L7
.L3:
    movl     $5, -8(%ebp)
.L2:
    cmpl     $10, -4(%ebp)
    jg       .L5
    jmp      .L4
.L7:
    nop
.L4:
    movl     $0, %eax
    leave
    ret
```

```
int main (void)
{
    int a, b;
    while ( a > 10)
    {
        if (a > b)
            break;
        b = 5;
    }
    return 0;
}
```

```

    jmp .L2
.L5:
    movl    -4(%ebp), %eax
    cmpl    -8(%ebp), %eax
    jg      .L7
.L3:
    movl    $5, -8(%ebp)
.L2:
    cmpl    $10, -4(%ebp)
    jg      .L5
    jmp     .L4
.L7:
    nop
.L4:
    movl    $0, %eax
    leave
    ret

```

```
int main (void)
{
    int a, b;
    while ( a > 10)
    {
        if (a > b)
            continue;
        b = 5;
    }
    return 0;
}
```

```

    jmp .L2
.L4:
    movl    -4(%ebp), %eax
    cmpl    -8(%ebp), %eax
    jg      .L6
.L3:
    movl    $5, -8(%ebp)
    jmp     .L2
.L6:
    nop
.L2:
    cmpl    $10, -4(%ebp)
    jg      .L4
    movl    $0, %eax
    leave
    ret

```

Guilherme R. Cardim e Ruben de Best

```
int main (void)
{
    int a, b;
    while ( a > 10)
    {
        if (a > b)
            goto nome;
        b = 5;
    }
nome:
    return 0;
}
```

```

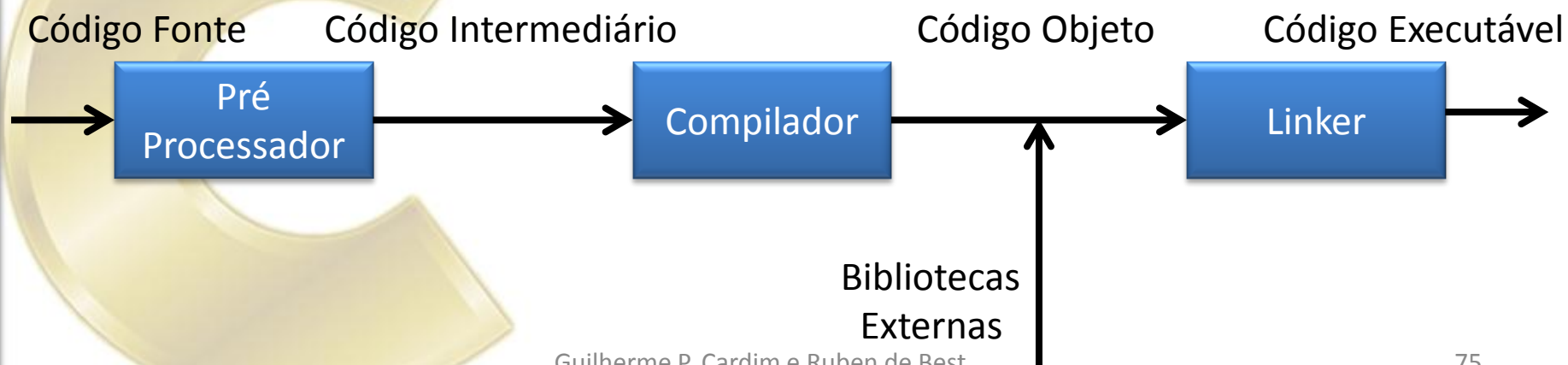
    jmp .L2
.L5:
    movl    -4(%ebp), %eax
    cmpl    -8(%ebp), %eax
    jg      .L7
.L3:
    movl    $5, -8(%ebp)
.L2:
    cmpl    $10, -4(%ebp)
    jg      .L5
    jmp     .L4
.L7:
    nop
.L4:
    movl    $0, %eax
    leave
    ret

```



Processo de Compilação

- O processo de Compilação em C envolve três passos principais:
 - Pré Processador;
 - Compilação;
 - Linkagem.





Pré Processador

- É responsável por processar as diretivas da linguagem. As principais são:
 - #include
 - #define
 - #undef
 - #ifdef
 - #ifndef
 - #if
 - #else
 - #elif
 - #endif



Pré Processador (...)

- As diretivas são processadas nesta etapa e geram um novo código para o compilador.

```
#define MAX 100
```

```
int main (void)
{
    int i;
    for(i=0;i<MAX;i++);
    return 0;
}
```

Pré Processador

```
int main (void)
{
    int i;
    for(i=0;i<100;i++);
    return 0;
}
```

Testes efetuado com o compilador gcc 4.6.3

```
#define max(a,b) ( a > b ? a : b)
```

```
int main (void)
{
    int x=1, y=0;

    printf("%d ",max(x++,y++));
    printf("\n%d %d\n",x,y);

    return 0;
}
```

Pré
Processador

```
int main (void)
{
    int x=1, y=0;

    printf("%d ",( x++ > y++ ? x++ : y++));
    printf("\n%d %d\n",x,y);

    return 0;
}
```



Compilador

- Recebe o código intermediário e gera as instruções de máquina em um código objeto.
- Trata cada arquivo fonte como uma unidade de compilação.
- Na verdade ele gera um código Assembly e logo em seguida executa um montador Assembler para gerar o código de máquina.

Compilador (...)



Código Intermediário

```
int main (void)
{
    int i;
    for(i=0;i<100;i++);
    return 0;
}
```

Assembly

Código Assembly

```
main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $0, -4(%ebp)
    jmp      .L2
.L3:
    addl     $1, -4(%ebp)
.L2:
    cmpl     $99, -4(%ebp)
    jle      .L3
    movl     $0, %eax
    leave
    ret
```

Código Objeto – Ling. de Máquina

```
00100101100101
01010011101010
01010101111011
```

Montador



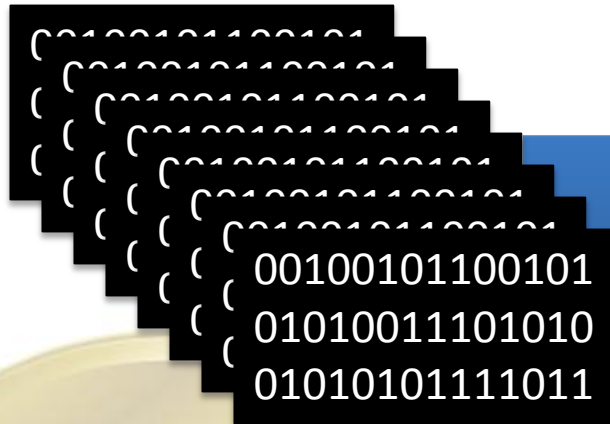
Linker

- Reune todos os códigos objetos em um único arquivo executável;
- Substitui todas as chamadas de funções e acessos a variáveis ao seu real endereço na memória;
- Organiza cada função e variável dentro do espaço da memória.

Linker (...)



Códigos Objeto



Linker

Arquivo Executável



```
void proc();  
  
int main(int argc, char* argv[])  
{  
    proc();  
  
    return 0;  
}
```

error LNK2019: unresolved external symbol _proc referenced in function _main



Considerações Finais

- Linguagem robusta;
 - Permite desenvolver aplicações para praticamente todos os tipos de problemas;
 - Permite a manipulação direta de memória;
- Rápida;
 - Características de baixo nível permitindo performances semelhantes ao Assembly;
- Flexível;
 - Altamente sujeita a erros por parte do programador.



Bibliografia

- <http://www.vivaolinux.com.br/artigo/Tratamento-de-excecoes-na-linguagem-C>
- http://pt.wikipedia.org/wiki/Tratamento_de_exce%C3%A7%C3%A3o
- http://www.icmc.usp.br/~luisc/download/fundamentos/Linguagem_Engenharia/AVALIACAO_DA_LINGUAGEM.pdf
- <http://www.cin.ufpe.br/~jrpn/arquivos/5%BA%20Periodo/Paradigmas/Aulas/Topico%202%20-%20Linguagens%20de%20Programa%E7%E3o%20Conceitos%20B%E1sicos.pdf>
- http://www.spectrum.eti.br/news/files/projeto_graduacao.pdf
- http://www.claudiorodolfo.com/ftc/plp/aula_01.pdf
- http://pt.wikibooks.org/wiki/Programar_em_C/Controle_de_fluxo#Saltos_incondicionais:_goto
- <http://pontov.com.br/site/cpp/61-aprendendo-o-c/280-parte-4-estruturas-de-controle>



Bibliografia (...)

- <http://www.mtm.ufsc.br/~azeredo/cursoC/aulas/c480.html>
- <http://www.estig.ipbeja.pt/~rmcp/estig/2002/1s/lp1/teorica/Processo%20de%20Compilacao.pdf>
- <http://www.pontov.com.br/site/cpp/46-conceitos-basicos/95-compilacao-de-programas-cc>
- http://pt.wikibooks.org/wiki/Programar_em_C%2B%2B/Compila%C3%A7%C3%A3o
- <http://www.lrc.ic.unicamp.br/~luciano/courses/mc202-2s2009/pre-processor.pdf>
- http://pt.wikibooks.org/wiki/Programar_em_C/Pr%C3%A9-processor#Usos_comuns_das_diretivas
- <http://www.mtm.ufsc.br/~azeredo/cursoC/aulas/c270.html>
- <http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/pointer.html>
- <http://stackoverflow.com/questions/2515598/push-ebp-movlesp-ebp>



Bibliografia (...)

- <http://stackoverflow.com/questions/2748995/c-struct-memory-layout>
- http://en.wikipedia.org/wiki/Data_structure_alignment
- <http://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/SYNTAX/enum.html>
- Kernighan, B. W. **C Linguagem de Programação Padrão ANSI**. Editora Campus 1989. Ed 2.
- Damas, L. **Linguagem C**. LTC 1999. Ed 10.
- Sebesta, R. W. **Conceitos de Linguagens de Programação**. Bookman 2003. Ed 5.
- Schildt, H. **C Completo e Total**. Makron Books 1996. Ed 3.
- GARCIA, R. E. **Linguagem de Programação**. Notas de Aula, Unesp-FCT 2012.